Performance of Parallel Two-Pass MDL Context Tree Algorithm

Nikhil Krishnan and Dror Baron
Department of Electrical and Computer Engineering
North Carolina State University; Raleigh, NC 27695, USA
Email: {nkrishn, barondror}@ncsu.edu

***Abstract*—Computing problems that handle large amounts of data necessitate the use of lossless data compression for efficient storage and transmission. We present numerical results that showcase the advantages of a novel lossless universal data compression algorithm that uses parallel computational units to increase the throughput with minimal degradation in the compression quality. Our approach is to divide the data into blocks, estimate the minimum description length (MDL) context tree source underlying the entire input, and compress each block in parallel based on the MDL source. Numerical results from a prototype implementation suggest that our algorithm offers a better trade-off between compression and throughput than competing universal data compression algorithms.**

***Index Terms*—big data, distributed computing, minimum description length, parallel algorithms, redundancy, two-pass code, universal data compression.**

I. Introduction

A. Motivation

The emergence of distributed cloud computing and big data problems raises new challenges in data storage and communication. In such distributed computing settings, the data may be processed remotely in clusters and the results are streamed to the end user through a network. The use of data streaming in big data problems makes it imperative to use fast lossless data compression algorithms whose primary features include good compression quality and high throughput.

Some applications of fast compression include internet backbone data compression and compression in high volume data generation applications such as scientific computing. Data can be compressed rapidly near the source of data generation, and can be transmitted through band limited channels. This compression scheme can reduce energy consumption and bandwidth requirements of the network.

Several techniques are available to improve the throughput, such as hardware acceleration [1], algorithmic approximations, and computer architecture optimizations [2–5]. Although these acceleration, approximation, and optimization techniques may accelerate compression, there are many systems where these do not suffice either due to limited speed up or poor compression quality. Ultimately, in order for lossless compression to become appealing for a broader range of applications, we must concentrate more on efficient new algorithms.

With the availability of cheap multi core processors such as graphics processing units (GPUs), parallelization is a possible direction for fast compression algorithms. Dividing the data into independent blocks to compress results in loss of compression quality [6]. Previous works that tried to address the loss in compression due to parallelization have limitations such as excess coding length above entropy rate [6], high time complexity [7], and not supporting scalable data rates [8].

We have introduced a parallel two-pass minimum description length (PTP-MDL) algorithm [9, 10] that divides the original input of length N into B blocks, and compresses all the blocks in a cooperative way. This PTP-MDL algorithm has a speed up that is linear in B without degrading the compression quality by sharing information across B blocks. PTP-MDL incorporates Rissanen's MDL principle [11], and the information sharing ideas of Beirami and Fekri [12]. The PTP-MDL algorithm has the useful property of random access [13], where any part of the compressed file can be decompressed without decompressing the entire file.

B. Contributions

This paper showcases numerical results that highlight the advantages of the PTP-MDL algorithm [9]. We compare the compression and throughput as a function of the number of parallel computational units available. Numerical results show that the PTP-MDL algorithm provides a better trade-off between compression and throughput, which makes this algorithm attractive for big data problems.

The remainder of the paper is organized as follows. We review relevant preliminary material on data compression, the PTP-MDL algorithm, and its key properties in Section II. Comprehensive numerical results on real data are presented in Section III.

II. Background and Algorithm

A. Data compression background

Universal data compression: Lower bounds on the redundancy, which is the excess coding length over the entropy, serve as benchmarks for compression quality.

This work was supported in part by the National Science Foundation under Grant CCF-1217749 and in part by the U.S. Army Research Office under Grant W911NF-04-D-0003.

Consider length-N sequences x generated by a stationary ergodic source over a finite alphabet $\mathcal{X}$, i.e., $x \in \mathcal{X}^N$. Rissanen [14] proved that, for universal coding of independent and identically distributed (i.i.d.) sequences, the worst case redundancy (WCR) is at least $\frac{|\mathcal{X}|-1}{2}\log(N)+O(1)$ bits, where $|\mathcal{X}|$ denotes the cardinality of $\mathcal{X}$, and logarithms are base-2. Because i.i.d. models are too simplistic for modeling "real-world" inputs, we use tree sources instead.

Tree sources: Let x_i^j denote the sequence $x_i, x_{i+1}, \ldots, x_j$ where $x_k \in \mathcal{X}$ for $i \leq k \leq j$. Let $\mathcal{X}^*$ denote the set of finite-length sequences over $\mathcal{X}$. Define a context tree source $\{\mathcal{S}, \Theta\}$ [15] as a finite set of sequences called states $\mathcal{S} \subset \mathcal{X}^*$ that is complete and proper [15, p.654], and a set of conditional probabilities $\Theta = \{p(\alpha|s) : \alpha \in \mathcal{X}, s \in \mathcal{S}\}$. We say that s generates symbols following it. Because $\mathcal{S}$ is complete and proper, the sequences of $\mathcal{S}$ can be arranged as leaves on an $|\mathcal{X}|$-ary tree [16]; the unique state s that generated x_i can be determined by entering the tree at the root, first choosing branch x_{i-1}, then branch x_{i-2}, and so on, until some leaf s is encountered. Let $D \triangleq \max_{s\in\mathcal{S}}|s|$ be the maximum context depth that we allow for the source. Then the string x_{i-D}^{i-1} uniquely determines the current state s; the previous symbols x_{i-L}^{i-1} ($L \leq D$) that uniquely determine the current state s are called the context, and L is called the context depth for state s.

Semi-predictive compression: Consider a tree source structure $\mathcal{S}$ whose explicit description requires $l_{\mathcal{S}}$ bits, and denote the probability of the input sequence x conditioned on the tree source structure $\mathcal{S}$ by $p_{\mathcal{S}}(x)$. Using $\mathcal{S}$, the coding length required for x is $l_{\mathcal{S}} - \log(p_{\mathcal{S}}(x))$. Let the MDL tree source structure $\widehat{\mathcal{S}}$ be the tree structure that provides the shortest description of the data, i.e.,

$$\widehat{\mathcal{S}} \triangleq \arg\min_{\mathcal{S}\in\mathcal{C}} \left\{l_{\mathcal{S}} - \log(p_{\mathcal{S}}(x))\right\},$$

where $\mathcal{C}$ is the class of tree source models being considered. The semi-predictive approach [17–19] processes the input x in two passes. Pass I first estimates $\widehat{\mathcal{S}}$ by context tree pruning, which minimizes the coding length. The structure of $\widehat{\mathcal{S}}$ is then encoded explicitly. Pass II uses $\widehat{\mathcal{S}}$ to encode the sequence x sequentially, where the parameters $\widehat{\Theta}$ are estimated while encoding x. The decoder first determines $\widehat{\mathcal{S}}$, and afterwards uses it to decode x sequentially from the two-pass code.

Two-pass compression: In contrast to the semi-predictive approach, which only encodes the structure of $\widehat{\mathcal{S}}$ in the Pass I, our two-pass approach also encodes parameter values $\widehat{\Theta}$. Despite the parallel nature of our algorithm, it incurs a single redundancy term for lack of knowledge of the parameters in Pass I instead of B redundancy terms in Pass II.

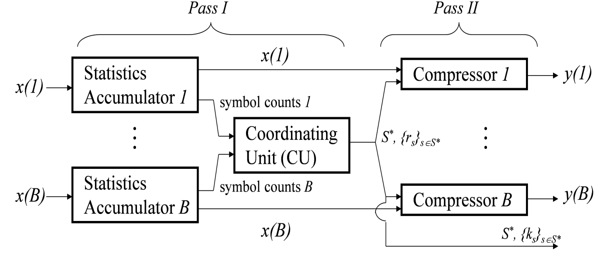

Figure 1: Block diagram of the PTP-MDL encoder.

B. *PTP-MDL algorithm*

In order to keep the presentation simple, we restrict our attention to a binary alphabet, i.e., $\mathcal{X} = \{0, 1\}$; the generalization to non-binary alphabets is straightforward.

Algorithm description: A block diagram of the PTP-MDL encoder is shown in Figure 1.

1) Pass I: In Pass I, the PTP-MDL encoder employs B computational units called parallel units (PUs) that work in parallel to accumulate statistical information on B blocks in $O(N/B)$ time, and a coordinating unit (CU) that controls the PUs and computes the MDL source estimate $\{\widehat{\mathcal{S}}, \widehat{\Theta}\}$.

Without loss of generality, we assume $N/B \in \mathbb{Z}^+$. Define the B blocks as $x(1) = x_1^{N/B}, x(2) = x_{N/B+1}^{2N/B}, \ldots, x(B) = x_{N-N/B+1}^{N}$. Parallel unit b, where $b \in \{1, \ldots, B\}$, first computes for each depth-D context s the block symbol counts $n_s^\alpha(b)$, which are the number of times the symbol $\alpha \in \{0, 1\}$ is generated by s in $x(b)$,

$$n_s^\alpha(b) \triangleq \sum_{i=(b-1)(N/B)+D+1}^{b(N/B)} 1_{\left\{x_{i-D}^{i}=s\alpha\right\}}, \alpha \in \mathcal{X},$$

where $s\alpha$ denotes concatenation of s and α, and $1_{\{\cdot\}}$ is the indicator function.

Having received the block symbol counts $n_s^\alpha(b)$ from the PUs, the CU computes the symbol counts generated by state s in the entire sequence x, $n_s^\alpha = \sum_{b=1}^{B} n_s^\alpha(b), \alpha \in \mathcal{X}$. The CU can then compute the maximum likelihood (ML) parameter estimates of $p(1|s)$ and $p(0|s)$, which are defined as $\theta_s \triangleq \theta_s^1 = \frac{n_s^1}{n_s^0+n_s^1}$ and $\theta_s^0 = 1 - \theta_s^1$, respectively. The ML parameter estimates for each state s are quantized into one of $O(\sqrt{N})$ representation levels based on Jeffreys' prior such that each bin has the same probability mass [20]. The representation levels and bin edges are computed using a closed form quantizer [20]. The bin index and representation level for state s are denoted by k_s and r_s, respectively. Denoting the quantized ML estimate of θ_s^α by $\widehat{\theta}_s^\alpha$, we have $\widehat{\theta}_s^1 = r_s$ and $\widehat{\theta}_s^0 = 1 - r_s$.

For each state s such that $|s| < D$, the CU either retains the children states $0s$ and $1s$ in the MDL source, or prunes them and only retains s, whichever results in

a shorter coding length. Details of the pruning decision appear in [9].

At the end of Pass I, the CU has computed the MDL structure estimate $\widehat{\mathcal{S}}$. If $s \in \widehat{\mathcal{S}}$, then the first part of the two-pass code for symbols generated by s consists of encoding k_s with $\log(K_s)$ bits. The WCR using this quantization approach is 1.047 bits per state above Rissanen's redundancy bound [10, 14, 20].

2) Pass II: In Pass II, which implements the second part of the two-pass code, each PU b encodes its block $x(b)$ sequentially. For each symbol $x_i(b)$, PU b first determines the generator state $G_i(b)$, the state s that generated the symbol $x_i(b)$. PU b then assigns $x_i(b)$ a probability $\widehat{p}(x_i(b)) \triangleq \widehat{\theta}_{G_i(b)}^{x_i(b)}$ according to the parameters that were estimated by the CU in Pass I, and sequentially feeds the probability assignments to an arithmetic encoder [21].

3) Decoder: The structure of the decoder is similar to that of Pass II. The approximated MDL source structure $\widehat{\mathcal{S}}$ and quantized parameters $\widehat{\Theta}$ are first derived from the first part of the two-pass code (sub-section II-B1). Then, the B blocks are decompressed by B decoding blocks. In decoding block b, each symbol $x_i(b)$ is sequentially decoded by determining $G_i(b)$, assigning a probability to $x_i(b)$ based on the parameter estimates, and applying an arithmetic decoder [21].

Key properties: The following two properties summarize the theoretical performance of the PTP-MDL algorithm. First, the redundancy of the PTP-MDL algorithm is upper bounded by $B\left[\log\left(\frac{N}{B}\right)+2\right] + \frac{|\widehat{S}|}{2}\left[\log(N)+O(1)\right]$ [9], where the $\log\left(\frac{N}{B}\right)$ term is for encoding the first $D=\log\left(\frac{N}{B}\right)$ bits in each block due to insufficient context depth. Second, with computations performed with $\log(N)$ bits of precision defined as $O(1)$ time and $D \leq \log(N/B)$, the PTP-MDL encoder and decoder each require $O(N/B)$ time. The sub-routines in the algorithm for achieving $O(N/B)$ time performance are detailed in [10].

III. Numerical Results

Having described the PTP-MDL algorithm, we have set the stage to describe our numerical results with a prototype implementation. After surveying our simulation setting, we compare the compression ratio and the throughput of the PTP-MDL algorithm with two types of algorithms: (*i*) high quality compressors; and (*ii*) fast compression algorithms. The PTP-MDL algorithm offers competitive performance with both types of algorithms. Finally, we show the trade-off between compression ratio and throughput of the PTP-MDL algorithm as as a function of B.

Compression algorithm	Compression ratio (bits/byte)			Average Throughput (Mbps)
	E.coli	bible.txt	world 192.txt	
LZ77a (32KB)	2.35	2.32	2.32	31
LZ77b (4MB)	2.27	1.93	1.72	36
BWT	2.16	1.67	1.58	36
NanoZip	1.97	1.42	1.25	84
Gipfeli	6.00	7.49	7.27	826
LZ4	5.45	4.14	3.98	1515
Snappy	3.73	3.93	4.02	2025
PTP-MDL (B=1)	1.98	2.20	2.58	2
-"- (B=10)	1.99	2.39	2.92	16
-"- (B=100)	1.99	2.59	3.25	135
-"- (B=1000)	2.01	3.10	3.79	952

Table I: Performance comparison for different compressors.

A. Simulation setting

We have a serial (non-parallel) implementation in C++, which serves as a prototype that can allow to evaluate anticipated performance of a GPU implementation, which is ongoing work. The algorithm treats any data as a binary bit stream; in future work, we plan to apply techniques that exploit the byte nature of real data [22]. Although the parallel parts of the algorithm run sequentially in the current implementation, we give a predicted time performance as

$$t_{\text{est}}(B) = t_{\text{s}} + \frac{t_{\text{sp}}}{\eta B}, \quad (1)$$

where $t_{\text{est}}(B)$ is the estimated time for executing the algorithm using B PUs, t_{s} is the time required to execute the serial part of the algorithm, t_{sp} is the sequential time required for executing the parallel part of the algorithm, and $\eta \in [0,1]$ is the efficiency of parallelization.

The compression ratio γ is defined as the average number of output bits in the compressed data required to represent one input byte of uncompressed data. Note that the lower the γ, the better the compression. The throughput, $\mu = \frac{N}{t_{\text{est}}(B)}$ is measured in megabits per second (Mbps). We assume $\eta = 0.2$ in our simulations to provide a conservative estimate of the throughput.

B. PTP-MDL vs. other algorithms

We compare the compression ratio and throughput of several different algorithms for files E.coli (4.42MB), bible.txt (3.85MB), and world192.txt (2.85MB) taken from the large Canterbury corpus (http://corpus.canterbury.ac.nz/descriptions) in Table I and Figure 2. The memory requirements of the algorithm limit us to compressing files of several hundred MB on typical desktops ca. 2014. We were unable to compare to the other parallel schemes mentioned in Section I due to their implementations not being available.

The performance of the PTP-MDL algorithm is compared with high quality compressors such as Lempel-

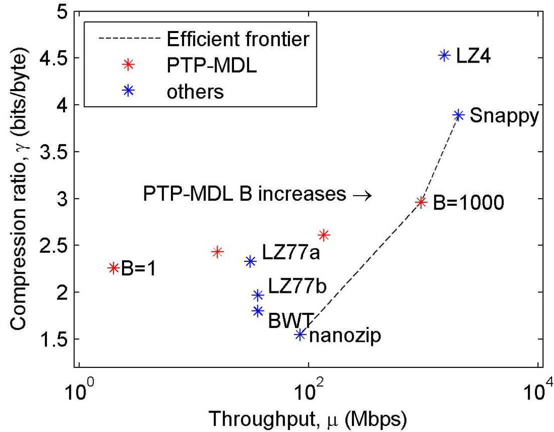

Figure 2: Compression ratio vs. throughput for different compressors.

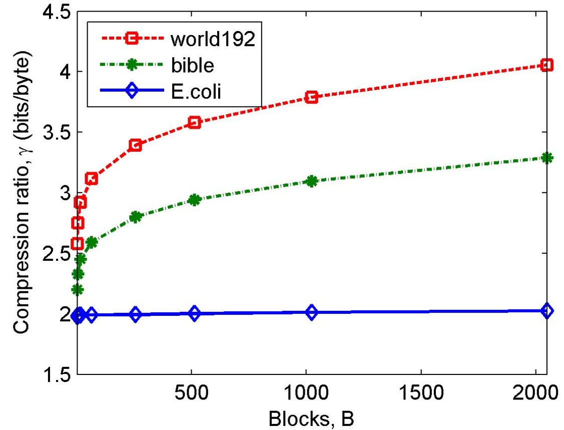

Figure 3: Compression ratio vs. number of blocks.

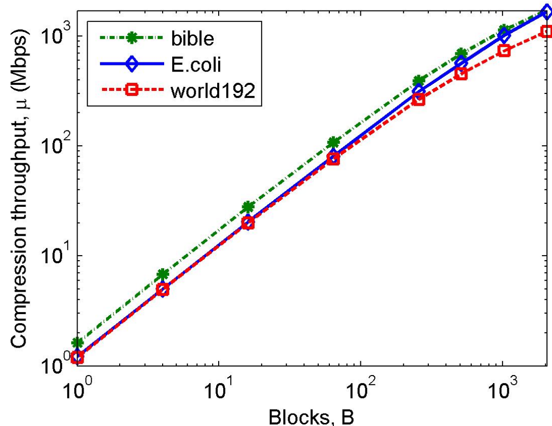

Figure 4: Throughput of compression vs. number of blocks.

Ziv coding [21] with a 32KB dictionary size (LZ77a), Lempel-Ziv coding with a 4MB dictionary size (LZ77b), the Burrows-Wheeler transform (BWT) [23], and the context-based NanoZip algorithm (http://nanozip.net/), as well as fast algorithms such as LZ4 [24], Gipfeli [2], and Snappy [3]. NanoZip gives the lowest compression ratios for all the files, and the PTP-MDL algorithm has a compression ratio very close to that of NanoZip for the E.coli file. The PTP-MDL algorithm has a competitive compression ratio although this algorithm is implemented for binary symbols, whereas the other algorithms are designed for 8 bit symbols. LZ4, Gipfeli and Snappy, which are speed optimized approximations of LZ77 [21], did not do well in compression ratio performance. In addition to a competitive compression ratio, as B increases, the throughput of the PTP-MDL algorithm is comparable to the throughputs of fast data compression algorithms such as LZ4 (≈ 1515 Mbps) and Snappy (≈ 2025 Mbps) (Table I), which are used in big data problems.

C. Compression ratio and throughput vs. B

PTP-MDL γ performance vs. B: Figure 3 illustrates the impact of the number of blocks B on the compression ratio γ. For a simple source such as E.coli, where D is small, the compression ratio increase with B is modest. However, there is a non-linear behavior for more complicated data such as English text. For bible.txt and world192.txt, the compression ratio γ deteriorates rapidly for small B. This deterioration could be due to the decrease in maximum depth available for the tree source given by $O(\log(N/B))$, which may impact the coding length as B increases.

PTP-MDL μ performance vs. B: Figure 4 illustrates the impact of the number of blocks B on the compression throughput μ. For small B, the speed up is linear, and as B increases, the speed up slows down. This trend is due to Amdahl's law [25] given in (1). When B is low, the fraction of serial execution time is insignificant compared to parallel execution time. However, when B is large, the parallel fraction of execution time is reduced, which introduces nonlinearity in the throughput μ for large values of B. The result for decompression throughput is similar to that of compression throughput, and thus omitted for brevity.

To summarize, our numerical results show that the compression ratio γ of our algorithm is comparable to existing universal data compressors for real data and the throughput μ scales well with the number of parallel units even for large B.

Acknowledgements

We thank Yoram Bresler and Mehmet Kıvanç Mıhçak for numerous discussions relating to this work; Frans Willems for the arithmetic code implementation; and Yanting Ma, Jin Tan, and Junan Zhu for their careful evaluation of the manuscript.

REFERENCES

[1] S. Arming, R. Fenkhuber, and T. Handl, "Data compression in hardware – the Burrows-Wheeler approach," in *IEEE Int. Symp. Des. Diagnostics Electron. Circuits Syst.*, Apr. 2010, pp. 60–65.

[2] R. Lenhardt and J. Alakuijala, "Gipfeli - high speed compression algorithm," in *Proc. Data Compression Conference (DCC)*, Apr. 2012, pp. 109–118.

[3] S. Gunderson, "Snappy, A fast compressor/decompressor," code.google.com/p/snappy/.

[4] L. M. Reinhold, "QuickLZ website," http://www.quicklz.com/.

[5] A. Hidayat, "FastLZ website," http://fastlz.org/.

[6] P. Franaszek, J. Robinson, and J. Thomas, "Parallel compression with cooperative dictionary construction," in *Proc. Data Compression Conf. (DCC)*, Mar. 1996.

[7] F. M. J. Willems, "Some challenges in source coding," in *Proc. 3rd ITG Conf. Source Channel Coding*, Jan. 2000, pp. 245–249.

[8] M. L. A. Stassen and T. J. Tjalkens, "A parallel implementation of the CTW compression algorithm," in *Proc. 22d Benelux Symp. Inf. Comm.*, May 2001, pp. 85–92.

[9] N. Krishnan, D. Baron, and M. K. Mıhçak, "A parallel two-pass MDL context tree algorithm for universal source coding," in *Proc. Int. Symp. Inf. Theory (ISIT)*, July 2014.

[10] D. Baron, "Fast parallel algorithms for universal lossless source coding," Feb. 2003, Ph.D. thesis, UIUC.

[11] J. Rissanen, "Modeling by shortest data description," *Automatica*, vol. 14, no. 5, pp. 465–471, Sept. 1978.

[12] A. Beirami and F. Fekri, "On lossless universal compression of distributed identical sources," in *Proc. Int. Symp. Inf. Theory (ISIT)*, July 2012, pp. 561–565.

[13] S. Kreft and G. Navarro, "LZ77-like compression with fast random access," pp. 239–248, March 2010.

[14] J. Rissanen, "Fisher information and stochastic complexity," *IEEE Trans. Inf. Theory*, vol. 42, no. 1, pp. 40–47, Jan. 1996.

[15] F. M. J. Willems, Y. M. Shtarkov, and T. J. Tjalkens, "The context tree weighting method: Basic properties," *IEEE Trans. Inf. Theory*, vol. 41, no. 3, pp. 653–664, May 1995.

[16] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, The MIT Press, Cambridge, MA, 2009.

[17] D. Baron and Y. Bresler, "An O(N) semipredictive universal encoder via the BWT," *IEEE Trans. Inf. Theory*, vol. 50, no. 5, pp. 928–937, May 2004.

[18] P. A. J. Volf and F. M. J. Willems, "A study of the context tree maximizing method," in *Proc. 16th Benelux Symp. Inf. Theory, Nieuwerkerk Ijsel, Netherlands*, May 1995, pp. 3–9.

[19] F. M. J. Willems, Y. M. Shtarkov, and T. J. Tjalkens, "Context-tree maximizing," in *Proc. Conf. Inf. Sci. Syst.*, Mar. 2000, pp. 7–12.

[20] D. Baron, Y. Bresler., and M. K. Mıhçak, "Two-part codes with low worst-case redundancies for distributed compression of Bernoulli sequences," in *Proc. Conf. Inf. Sciences Systems*, Mar. 2003.

[21] T. M. Cover and J. A. Thomas, *Elements of Information Theory*, New York, NY, USA: Wiley-Interscience, 2006.

[22] F.M.J. Willems, "The context-tree weighting method: Extensions," *IEEE Trans. Inf. Theory*, vol. 44, no. 2, pp. 792–798, Mar. 1998.

[23] M. Burrows and D.J. Wheeler, *A block-sorting lossless data compression algorithm*, 1994.

[24] "LZ4, Extremely Fast Compression algorithm," code.google.com/p/lz4/.

[25] Y. Solihin, *Fundamentals of Parallel Computer Architecture*, Solihin Publishing and Consulting LLC, 2009.